**什么是API？**

应用程序编程接口。（Application Program Interface）

整个JDK的类库就是一个javase的API。

每一个API都会配置一套API帮助文档。

Sun公司提前写好的类库就是API。（一般每一份API都会对应一份API帮助文档）

**String类**

String类中已经重写了equals方法和toString()方法，调用equals时会比较String类中具体的值是否相等。调用toString()方法时会输出字符串。

其实本质上String也只是一个普通的类。只不过Sun公司将其放入了源码中供开发者直接调用而已。

**数组**

数组在存放父类对象类型中的子类对象时，调用子类特有对象要向下转型，调用父类方法时可以直接调用。

**数组查找：二分法**

二分法查找是建立在排序好的数组中的进行的，没有排序好的数组不能用二分法查找。

二分法就是从中间（length/2）开始找。比较之后再从对应的数组中间开始找

**编程习惯**

可以先写伪代码，然后根据伪代码再写代码。

**异常**

抛出异常时可以抛出多个，中间用”，”隔开就行。

用throw抛出异常，称为“手动抛出异常”，就是直接new异常对象，他用于在语句中，一般自定义异常是手动抛出但并不会自己捕获，不然没有意义，抛出异常是为了将异常信息传递出去，所以一般用throws抛给上级。

throws自动抛出异常实在类名后面，可以跟多个。

JDK8之后的版本catch语句中的捕获异常可以用”|”（“或”）隔开，做到多个异常的处理。

关于throws和try...catch语句的选择：如果希望调用者来处理，就用throws，其他都用try...catch

异常的两个方法getMessage()和printStackTrace()，后者在实际开发中用的比较多。

重写（覆盖）之后的方法不能比重写之前的方法抛出更多（更宽泛）的编译时异常，可以更少。但运行时异常没有这个限制。getMessage()方法的返回值要用sout输出，但printStackTrace()不需要。

**IDEA注释快捷键及一些注意**

单行注释：”Ctrl+ /”

多行注释： ”Ctrl+Shift+ /”

文档注释：”输入/\*\* ,点击Enter”

IDEA中紫色代表实例变量

**final finally 和finalize**

final是一个关键字，可用修饰变量、类和方法，但final修饰的变量不可修改，修饰的类无法继承，修饰的方法无法覆盖

finally也是一个关键字。只能和try连用，用于处理异常的机制

finalize是Object类中的一个方法，所以他是一个标识符，但自从JDK9之后就不再使用了。由JVM中的GC垃圾回收器负责调用。

**集合**

1. 为什么Collection集合中的Set子接口下的HashSet和TreeSet接口new的是Map类型的数据结构？

答：原因就是底层仍然调用的是Map类型的数据结构，但是是将数据存到Map中的键的位置，Map集合的键就是Set集合。所以与普通的Map并不相同。

1. 由于contains方法底层调用了equals方法，所以放在集合中的元素要重写equals方法
2. Collection的Iteration方法中创建的迭代器创建时，其实相当于是对当前集合做一个快照，当集合结构发生改变，快照与集合本身不相同时，就比如用集合自身的remove方法删除元素时，集合就够就发生了改变，再用迭代器做任何操作时，都会出现同步异常：java.util.ConcurrentModificationException。但迭代器本身的方法在改变快照结构时也会同时改变集合结构，所以用迭代器方法删除元素时就没有问题。

**整理需要重写的几个方法：equals、hashCode、clone 、toString**

**以讲解equals 和 hashColde 为主，toString最常用但比较简单**

1. **Equals**

引用类型的数据在比较时都需要重写equals方法

Map集合中的key元素要重写hasCode方法

1. **hashCode**

不需要手动编写，一般和equals方法一起重写。IDEA可以自动生成。

Map集合中的key元素要重写hasCode方法

1. **CompareTo**

自定义泛型的TreeMap需要重写compareTo方法或设置比较器Comparator。

List类型集合若是要使用sort排序需要重写compareTo或设置比较器Comparator。（基本数据类型会自动装箱，所以因8种数据类型的包装类都重写了compareTo方法，所以可以直接比较）

1. **Clone（不是很重要）**

clone方法重写分为深克隆和浅克隆，深克隆是将一个对象包括他的成员变量一直划分到8个基本数据类型为止都要进行clone重写。如果不重写clone方法，那clone方法就是浅克隆，他只会将当前使用这个clone方法的对象克隆一份，但内部的成员变量和属性都指向的是原来被克隆对象的地址。

重写clone方法时，重写clone方法的类必须实现(implements)Cloneable接口，而克隆方法也必须抛出(throws)CloneNotSupportedException异常，在调用clone方法时也要抛出(throws)或者处理(try...catch)这个异常。

基本数据类型的clone重写IDEA可以直接添加且无需重写

引用数据类型的clone方法如下：

protected Object clone() throws CloneNotSupportedException {  
 CloneTest01 ct = (CloneTest01) super.clone();  
 ct.clonetest02 = (CloneTest02) clonetest02.clone();  
 return ct;  
}

这个有需要还得网上在看看，目前还不太理解。

1. **toString**

toString方法非常常用，如果直接输出实例化变量时，println方法会自动调用toString方法，返回的是对象的地址，不重写时toString输出格式为：变量名@内存地址

但toString本意就是输出对象的内容，所以重写也很频繁，根据需要重写。IDEA可自动生成。

**Java和windows的一些差别**

char在java中占两个字节，在Windows操作系统中占一个字节。

**序列化版本号**

如果程序中没有手动写出来序列化版本号，那么java虚拟机会根据编译了当前类之后形成的新的class文件自动生成序列化版本号。虚拟机在对比两个类是否相同时，会先比较类名，再比较类的序列化版本号。但自动生成的序列化版本号会在改变类的代码之后也改变，会形成新的序列化版本号，这样就会导致在使用ObjectInputStream反序列化对象时出现错误。因此，我们在写继承了Serializable接口的类时，建议手动添加一个序列化版本号，这个序列化版本号是long类型，数字可以随便写，但在项目中需要区分开来。例：Private static final serialVersionUID = 12312343L;

序列化版本号应用场景：十年前的类十年后添加属性，反序列化十年前的对象时。若是没有手动写序列化版本号，用ObjectInputStream反序列化十年前的对象时会报错：当前类的序列化版本号与在流中的序列化版本号不同。

IDEA工具也可以设置序列化版本号，但需要设置一下。

**属性配置文件(.properties)**

属性配置文件”#”是注释符号。属性配置文件中若是有重复的key值，那么最下面的这个key对应的value值会覆盖之前的key所对应的value值。

再key和value中间的等号“=”两边不建议加空格，低版本是取不出来的。但即使高版本可以取出，以后用到String框架时也可能会出现问题，所以最好是按照“key=value”的形式存储键值对。

**线程同步问题**

自己写的售票员卖票程序总结：

1. 首先要搞清楚线程是谁，对象是谁，第一次的程序设计没有搞清楚谁是对象谁是线程，其实线程就可以看作一个人，因为线程可以自己独立工作。所以我们创建的线程对应的应该是人本身。例如售票员售票系统，一个“销售地点”应该被看作一个对象类创建出来，而“销售员”在“销售地点”工作，这时“销售员”共享总票数，且有自己的买票统计，所以“销售员”是比较适合当线程的。

当这个问题解决后，思路就会清晰很多，最开始我的程序创建了销售员类且将线程当作是“销售地点”，这样的问题就是，我每多一个“销售员”，就必须创建一个“销售地点”来“存放”这个“销售员”，main方法中的程序就会非常冗余，而且，由于线程是工作台，因此总票数的设定很不方便，若是每个都设置了，那么这4个线程的相当于各自有各自的票数，总票数就不是统一的，所以我不得不将总票数设置为静态变量，才能让总票数共享。后来我发现可以将“工作地点”看作是一个对象，线程当作是“销售员”，这样我只需要创建一个“销售地点”，就可以用多个线程当作“销售员”来放到“销售地点”，减少了很多代码。

1. 其次是逻辑上的问题，不同的“销售员”应该有各自的销售票数，这些票数加起来应该是等于总票数，但是要将这个各自的售票数量各自统计，又让我非常烦恼。遇到这种问题，首先应该明确，多个线程共享的是方法区，独立的是run()中的方法（栈），所以每个线程各自的计数器设置在哪里很关键。首先要明确，若是将各自的计数器设置成类的成员变量时是肯定不行的，多个线程共享方法区和堆内存，所以成员变量在线程之间一定是继承关系，若是对成员变量进行修改操作，是会影响到下一个线程的执行的，所以成员变量作为线程各自的计数器肯定不合适。我自己研究出两种办法（在循环中）：第一种是将各自的计数器作为run方法调用的方法的实参传进去，然后在while方法中对计数器进行自加或自减，这是一种方法，原理就是每个线程运行自己的 run方法时都相当于初始化了一个为0的计数器，然后他的总数是共享的，计数器也不会被其他线程，因为其他线程也有自己的计数器，因此当while运行了多少次，这个计数器就会运行多少次。第二种和第一种区别不大，是将计数器放到run方法调用的方法中，原理差不多，但不能将初始值设置在while里面，只能放在外面，之后在while里面进行自加自减。也是一种不错的方法。
2. 最后是线程同步代码块synchronized的使用问题。首先如果不用这个同步代码块会出现卖出的数量大于总数量的问题，原因是：

买票的过程是：获取票数-->修改票数-->更新票数

多个线程在争夺CPU时间片时，在更新票数之前使用完了自己的时间片，就是总票数实际上已经减少了，但是数据还没有传过去，而其他线程就会获取到修改之前的数据进行操作，之后返回的数据可能与上一个线程的更新数据相同，假如一个循环减少一张票，那么这个过程实际上是减少了2张票，但原票数由于某个线程的数据延迟所以只减少了1票，那么就相当于总票数多了一票，这就是为什么卖出的数量比总数量多的原因。

而放在synchronized代码块中的代码就相当于是一个时间片，线程要是抢到足够时间片就执行完这里面的代码，要是抢不到足够的时间片就不执行这里面的代码，这样就保证了数据一定是在更新完之后才会被其他线程修改，这就是线程同步，也使数据安全性很高。简单的说，放在synchronized的小括号里面的对象，如果是方法中的变量，就相当于在每个线程中都有，他就是局部变量，是不共享的。如果是类中的成员变量，当只有一个类时，这个变量是这个类的属性，存放于堆内存中，是线程共享的。怎么说呢，synchronized小括号中存放的对象，只要是你所需要的线程同步所共享的，都可以让通向这个对象的线程同步。synchronized只是一个标识符。

**锁的问题**

被synchronized修饰，就说明会用到锁，没被synchronized修饰的与锁无关。

synchronized代码快锁的都是对象，称为对象锁。对象锁意思就是同一时间只能有一个线程访问这个对象的synchronized代码块，这个线程就获得了这个对象的锁，其他线程只能等他执行结束之后才能访问。

synchronized关键字修饰的实例方法，也是对象锁，若是有多个被synchronized修饰的方法，这个方法访问只能一个一个来，都需要等待上一个方法结束后，才能被某个访问，但没有被synchronized修饰的实例方法可以正常访问，因为访问这些方法不需要获取对象的对象锁。当然若是有多个对象，多个线程对应不同的对象，那么也不需要等，因为每一个对象都有自己的锁。

synchronized修饰的静态方法，是类锁，且静态方法是存放在方法区的，即使有多个对象，但他们的引用都是相同的，调用这个类中的任何有synchronized方法也需要排队调用，所以仍然需要等待。

**以后开发中，线程安全的选择问题**

Synchronized会让程序的执行效率降低，用户体验不好，系统的吞吐量降低。用户体验差。在不得已的情况下在使用线程同步机制。

第一种方案：使用局部变量代替实例变量和静态变量。

第二种方案：如果必须是实例变量，那么可以考虑创建多个对象，这样变量的内存就不共享了（一个线程对应1个对象，多个线程对应多个对象，对象不共享，就没有数据的安全问题）。

第三种方案：如果不能使用局部变量，且对象也不能创建多个，这个时候就只能选择synchronized线程同步机制了。

**关于JDK中自带的类加载器**

JDK中共有三类加载器：

启动类加载器：rt.jar

扩展类加载器：ext\\*.jar

应用类加载器：classpath

假如有这样一段代码：

String s = “ABC”;

代码在开始执行之前，会将所需要的类全部加载进JVM中，通过类加载器加载，看到以上代码，类加载器会找到String.class文件，找到就加载。

如何进行加载？

首先通过“启动类加载器”加载。

注意：启动类加载器专门加载:C:\Program Files\Java\jdk1.8.0\_101\lib\rt.jar中的class文件（但在JDK14上面并没有jre这个文件，然后我找了找也没找到rt.jar，推测应该肯

定是有这个文件的，但我没找到而已），rt.jar中都是JDK最核心的类库。

当“启动类加载器”加载不到时，

会通过“扩展类加载器”加载。

注意：扩展类加载器专门加载:C:\Program Files\Java\jdk1.8.0\_101\lib\ext\\*.jar文件

当“扩展类加载器”加载不到时，

会通过“应用类加载器”加载

注意：应用类加载器专门加载：classpath中的类。

Java中为了保证类加载的安全，使用了双亲委派机制

优先从启动类加载器中加载，这个称为“父”。“父”无法加载到，再从扩展类加载器中加载，这个称为“母”。双亲委派。如果都加载不到，才会考虑从应用类加载器中加载，直到加载到为止。

**Java的根路径（类路径）**

虽然一直说java的根路径是从src开始的，但若是用流读取java文件找不到是为什么？

因为java的真正的根路径（类路径）是当初设置的java项目存放地点的项目文件下的out文件下的文件，这里面存放的是已经编译过的class文件和其他类型的文件，java文件时不会存放在里面的，所以读取java文件会读取不到。